DOCKER

AGENDA

- Introduction to Container
- Docker Setup
- Docker Adhoc Commands
- Docker Manifest
- Building Image
- Docker Volumes, Linking, Ambassador Container
- Docker Compose
- Kubernetes
- Kubernetes Setup
- Kubernetes Basic Command

CONTAINER WORLD

WHAT IS CONTAINER

- What Problem Container Solves?
  - Ship software reliably from one computing environment to another
  - Shipping can be from local laptop to test environment or from staging to production or from a local datacenter to cloud environments
- How does Container solve that problem?
  - A container consists of entire runtime environment
    - Application artefact
    - All dependent libraries of application
    - Other third party binaries
    - Configuration files

Containerizing the app platform and its dependencies abstracts the differences in OS distributions and underlying infrastructure

WHAT IS CONTAINER

According to Wikipedia (and others):

> Docker is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating system-level virtualization on Linux.*

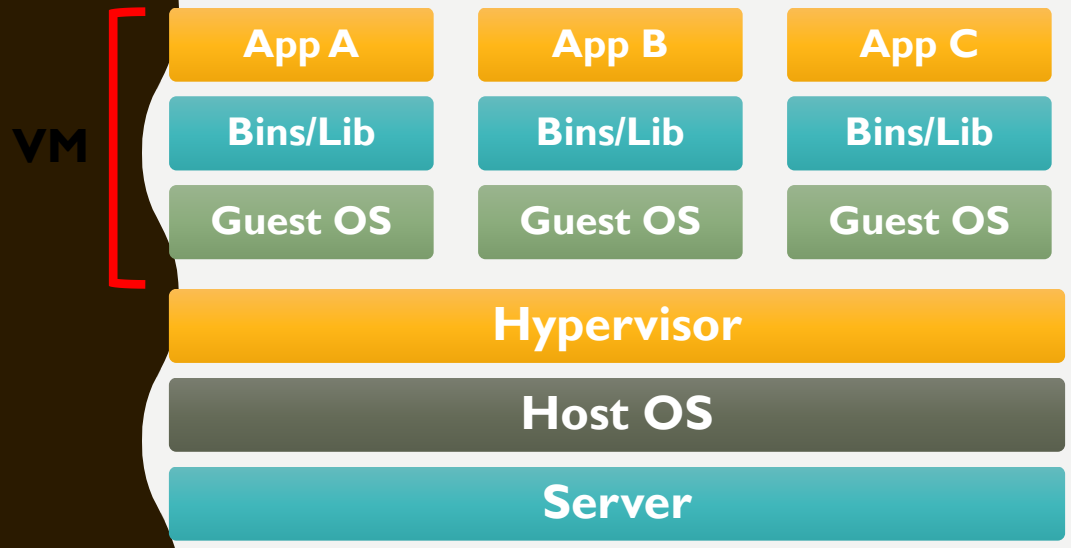
- It is a tool (or a set of tools) that packages up an application and all its dependencies in a "virtual container"
- It enables it to be run on **any** Linux system or distribution

Containerizing the app platform and its dependencies abstracts the differences in OS distributions and underlying infrastructure

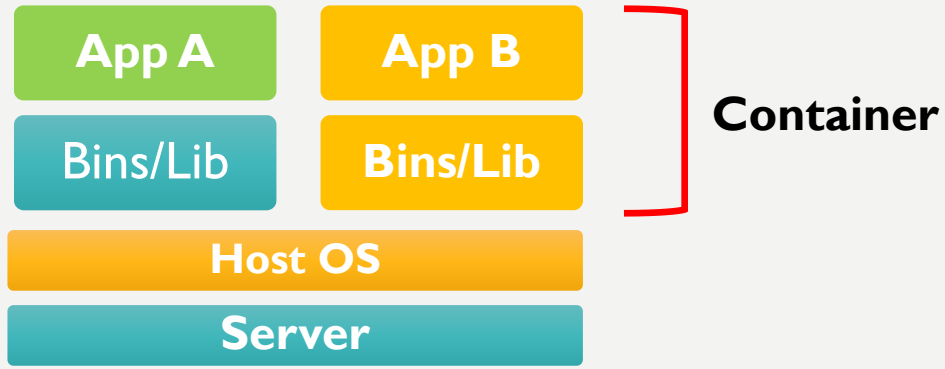
CONTAINER ARCHITECTURE

Container Architecture

Instead of virtualizing hardware, containers rest on top of a single Linux instance. This allows Docker to leave behind a lot of the bloated associated with a full hardware hypervisor. Here is a look at the architecture of each:



Containers are isolated, but share OS and, where appropriate, bins/libraries



DOCKER ARCHITECTURE

Architecture

- Docker is a client-server application where both the daemon and client can be run on the same system
- OR you can connect a Docker client with a remote Docker daemon.

Docker clients and daemons communicate via sockets or through a RESTful API.

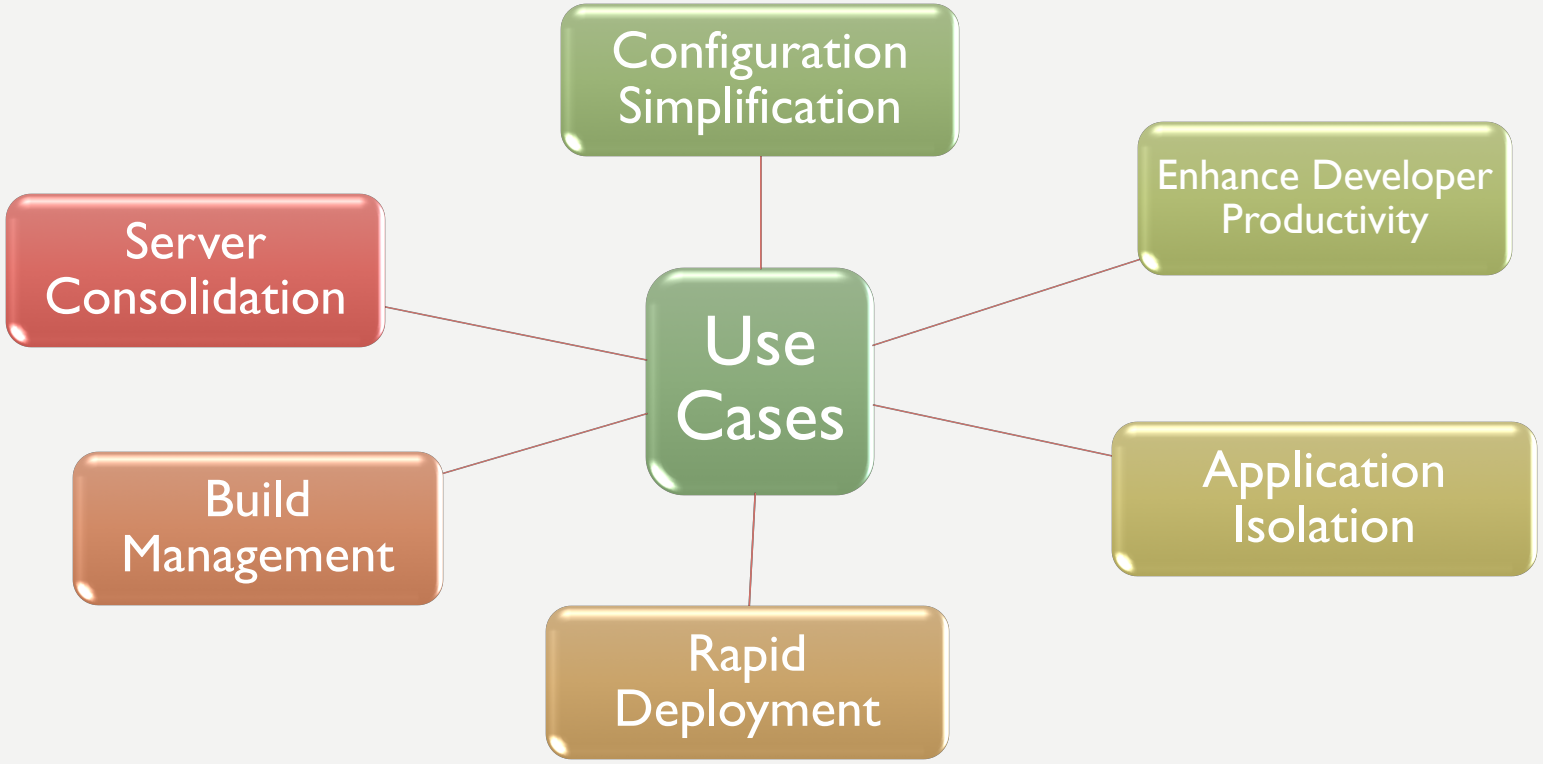
The main components of Docker are:

- Daemon
- Client
- Docker.io registry (Docker Hub)

SUMMARY

- Containerization makes the shipment of software easier
- Docker is most popular form of container
- Container abstracts the Applications with all its runtime dependencies
- Containers are becoming the standard unit of deployment
- Each Container Image
  - Code
  - Binaries
  - Configuration
  - Libraries
  - Frameworks
  - Runtime
- Developers and operators both love Containers

DOCKER – USE CASES



DOCKER SETUP

- Docker Installation
- Installation Verification
- Docker Hub
- Local Docker Repository

DOCKER INSTALLATION

- Go to the site https://docs.docker.com/engine/installation/
- Follow the Installation Instruction in the page mentioned above

Verify the Installation By

```
arun$ docker –version

Docker version 17.04.0-ce-rc2, build 2f35d73
```

DOCKER REPOSITORY

- Official Docker repository URL is https://hub.docker.com/
- Create a Docker Hub account which is free
- Docker Hub hosts official Images as well as images created by public
- You can host your own image in Docker Hub

DOCKER ADHOC COMMANDS

arun$ docker ps

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
5f0d5cd6c6fe	registry:2.6.0	"/entrypoint.sh /e..."	2 weeks ago	Up 3 days	0.0.0.0:5000->5000/tcp	dockerregistry_registry_1

arun$ docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
openshift/origin	latest	a204d456e2cd	4 days ago	635MB

arun$ docker search jenkins

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
jenkins	Official Jenkins Docker image	2731	[OK]	
stephenreed/jenkins-java8-maven-git	Automated build that provides a continuous...	56		[OK]

DOCKER ADHOC COMMANDS

```
arun$ docker pull jenkins/jenkins:latest
Using default tag: latest
latest: Pulling from library/jenkins
Digest: sha256:c0cac51cbd3af8947e105ec15aa4bcdbf0bd267984d8e7be5663b5551bbc5f4b
```

```
arun$ docker ps -a
CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS                      PORTS               NAMES
fa0dcd15bd9a        ubuntu                         "/bin/bash"              17 seconds ago      Exited (0) 16 seconds ago                       trusting_bohr
22ad7ee7627e        openshift/origin               "/usr/bin/openshif..."   4 days ago          Created                                         origin
7c74e2de1ae6        williamyeh/ansible:ubuntu16.04 "/bin/bash"              7 days ago          Exited (127) 7 days ago                         ansible
09e2a31a3f11        ubuntu                         "bash"                   9 days ago          Exited (0) 8 days ago                           ubuntu
```

DOCKER ADHOC COMMANDS

```
arun$ docker run nginx
```

```
arun$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
dac7234c593e	nginx	"nginx -g 'daemon ..."	18 seconds ago	Up 17 seconds	80/tcp, 443/tcp	**flamboyant_dubinsky**
5f0d5cd6c6fe	registry:2.6.0	"/entrypoint.sh /e..."	2 weeks ago	Up 3 days	0.0.0.0:5000->5000/tcp	dockerregistry_registry_1

```
arun$ docker stop flamboyant_dubinsky
```

```
arun$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
5f0d5cd6c6fe	registry:2.6.0	"/entrypoint.sh /e..."	2 weeks ago	Up 3 days	0.0.0.0:5000->5000/tcp	dockerregistry_registry_1

DOCKER ADHOC COMMANDS

- arun$ docker run –d nginx
- arun$ docker run –d –name my-nginx nginx

Port Mapping

```
arun$ docker run –d –p 8080:8080 nginx
```

Attaching Volumes

```
arun$ docker run –d –v /usr/arun:/app –p 8080:8080 nginx
```

DOCKER ADHOC COMMANDS

Entering the Container Shell

```
arun$ docker run –it –p 8080:8080 nginx bash

arun$ docker exec –it <container_id> bash
```

Removing Container

```
arun$ docker rm <container_id>
```

DOCKER MANIFEST - DOCKERFILE

Sample Dockerfile

```
FROM node:7
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY package.json /usr/src/app
RUN npm install
COPY . /usr/src/app
EXPOSE 4200
CMD ["npm", "start"]
```

DOCKER MANIFEST - DOCKERFILE

Create image based off of the official Node 7 image

```
FROM node:7
```

Create a directory where our app will be placed
RUN -

```
RUN mkdir -p /usr/src/app
```

Change directory so that our commands run inside this new dir
WORKDIR -

```
WORKDIR /usr/src/app
```

Copy dependency definitions
COPY -

```
COPY package.json /usr/src/app
```

DOCKER MANIFEST - DOCKERFILE

Install Dependencies

```
RUN npm install
```

Get all the code needed to run the app

```
COPY . /usr/src/app
```

Expose the port the app runs in

```
EXPOSE 4200
```

Start the app
CMD-

```
CMD ["npm", "start"]
```

DOCKER COMMANDS – BUILDING THE CONTAINER

Install Dependencies

```
RUN npm install
```

Get all the code needed to run the app

```
COPY . /usr/src/app
```

Expose the port the app runs in

```
EXPOSE 4200
```

Start the app
CMD-

```
CMD ["npm", "start"]
```

DOCKER COMMANDS – SAVING THE RUNNING CONTAINER

Install Dependencies

```
RUN npm install
```

Get all the code needed to run the app

```
COPY ./usr/src/app
```

Expose the port the app runs in

```
EXPOSE 4200
```

Start the app
CMD-

```
CMD ["npm", "start"]
```

DOCKER COMMANDS – DIFFING THE RUNNING CONTAINER

Install Dependencies

```
RUN npm install
```

Get all the code needed to run the app

```
COPY . /usr/src/app
```

Expose the port the app runs in

```
EXPOSE 4200
```

Start the app
CMD-

```
CMD ["npm", "start"]
```

BUILDING CONTAINER IMAGES

BUILDING CONTAINER IMAGES

- Dockerfile
- Docker build
- Building a base image from the Dockerfile
- Environmental Variables
- Putting it all together

DOCKERFILE INTRODUCTION

A Dockerfile is simply a plain text file that contains a set of user-defined commands, which when executed by the docker image build command--which we will look at next--assemble a container image.

```
FROM alpine:latest
LABEL maintainer="Russ McKendrick <russ@mckendrick.io>"
LABEL description="This example Dockerfile installs NGINX."

RUN apk add --update nginx && \
rm -rf /var/cache/apk/* && \
mkdir -p /tmp/nginx/

COPY files/nginx.conf /etc/nginx/nginx.conf
COPY files/default.conf /etc/nginx/conf.d/default.conf
ADD files/html.tar.gz /usr/share/nginx/

EXPOSE 80/tcp

ENTRYPOINT ["nginx"]
CMD ["-g", "daemon off;"]
```

DOCKERFILE IN DEPTH

- Commands used in the Dockerfile example in order in which they appear
  - FROM
  - LABEL
  - RUN
  - COPY and ADD
  - EXPOSE
  - ENTRYPOINT and CMD
  - Other Dockerfile Commands

FROM

The FROM command tells Docker which base you would like to use for your image;

In the example as shown above, we are using Alpine Linux, so we simply have to put the name of the image and also the release tag we wish to use.

```
FROM alpine:latest
```

LABEL

The LABEL command can be used to add extra information to the image. This information can be anything from a version number to a description.

```
$ docker image inspect <IMAGE_ID>
```

Alternatively, you can use the following to filter just the labels:

```
$ docker image inspect -f {{.Config.Labels}} <IMAGE_ID>
```

RUN

The RUN command is where we interact with our image to install software and run scripts, commands, and other tasks.

```
$ RUN apk add --update nginx && rm -rf /var/cache/apk/* && mkdir -p /tmp/nginx/
```

The first of our three commands, apk add --update nginx, installs NGINX using Alpine Linux's package manager; we are using the && operator to move on to the next command if the previous command was successful

```
RUN apk add --update nginx
RUN rm -rf /var/cache/apk/*
RUN mkdir -p /tmp/nginx/
```

COPY & ADD

At first glance, COPY and ADD look like they are doing the same task; however, there are some important differences. The COPY command is the more straightforward of the two:

```
COPY files/nginx.conf /etc/nginx/nginx.conf
COPY files/default.conf /etc/nginx/conf.d/default.conf
```

As you have probably guessed, we are copying two files from the files folder on the host we are building our image on. The first file is nginx.conf, which contains a basic NGINX configuration file:

```
ADD files/html.tar.gz /usr/share/nginx/
```

The ADD command can also be used to add content from remote sources:

```
ADD http://www.myremotesource.com/files/html.tar.gz /usr/share/nginx/
```

EXPOSE

The EXPOSE command lets Docker know that when the image is executed, then the port and protocol defined will be exposed at runtime.

```
EXPOSE 80/tcp
```

This command does not map the port to the host machine, but instead opens the port to allow access to the service on the container network. For example, in our Dockerfile, we are telling Docker to open port 80 every time the image runs:

ENTRYPOINT & CMD

The benefit of using ENTRYPOINT over CMD, which we will look at next, is that you can use them in conjunction with each other. ENTRYPOINT can be used by itself, but remember that you would want to use ENTRYPOINT by itself only if you wanted to have your container be executable

```
ENTRYPOINT ["nginx"]
CMD ["-g", "daemon off;"]
```

What this means is that whenever we launch a container from our image, the nginx binary is executed as we have defined that as our ENTRYPOINT, and then whatever we have as the CMD is executed, giving us the equivalent of running the following command:

```
$ nginx -g daemon off;
```

```
$ docker container run -d --name nginx dockerfile-example -v
```

OTHER DOCKERFILE COMMANDS

- USER
- WORKDIR
- ONBUILD
- ENV

USER

- The USER instruction lets you specify the username to be used when a command is run. The USER instruction can be used on the RUN instruction, the CMD instruction, or the ENTRYPOINT instruction in the Dockerfile. Also, the user defined in the USER command has to exist or your image will fail to build. Using the USER instruction can also introduce permission issues, not only on the container itself but also if you mount volumes.

WORKDIR

- The WORKDIR command sets the working directory for the same set of instructions that the USER instruction can use (RUN, CMD, and ENTRYPOINT). It will allow you to use the CMD and ADD instructions as well.

ONBUILD

- The ONBUILD instruction lets you stash a set of commands that will be used when the image is used again as a base image for a container. For example, if you want to give an image to developers and they all have different code they want to test, you can use the ONBUILD instruction to lay the groundwork ahead of the fact of needing the actual code. Then, the developers will simply add their code to the directory you tell them and, when they run a new docker build command, it will add their code to the running image.
  The ONBUILD instruction can be used in conjunction with the ADD and RUN instructions. Look at this example:
- **ONBUILD RUN apk update && apk upgrade && rm -rf /var/cache/apk/*** This would run an update and package upgrade every time our image is used as a base for another image.

ENV

- The ENV command sets environment variables within the image both when it is built and when it is executed. These variables can be overridden when you launch your image.

DOCKERFILE BEST PRACTICES

- Now that we have covered Dockerfile instructions, let's take a look at the best practices of writing our own Dockerfiles:
- You should try to get into the habit of using a .dockerignore file. We will cover the .dockerignore file in the next section; it will seem very familiar if you are used to using a .gitignore file. It will essentially ignore the items you have specified in the file during the build process.
- Remember to only have one Dockerfile per folder to help you organize your containers.
- Use version control for your Dockerfile; just like any other text-based document, version control will help you move forward, but only backward as necessary.
- Minimize the number of packages you need per image. One of the biggest goals you want to achieve while building your images is to keep them as small as possible. Not installing unnecessary packages will greatly help in achieving this goal.
- Execute only one application process per container. Every time you need a new application, it is best practice to use a new container to run that application in. While you can couple commands into a single container, it's best to separate them. Keep things simple; over complicating your Dockerfile will add bloat and also potentially cause you issues further down the line.
- Learn by example, Docker themselves have quite a detailed style guide for publishing the official images they host on Docker Hub, documented at https://github.com/docker-library/official-images/.

DOCKER COMPOSE

- Docker makes multi container system possible
  - But cumbersome
  - Only interface is command line

Docker-Compose

- Specify a single or multi-container system declaratively

```
version: '2'
services:
  angular:
    build: angular-client
    ports:
      - "4200:4200"
  express:
    build: express-server
    ports:
      - "3000:3000"
    links:
      - database

  database:
    image: mongo
    ports:
      - "27017:27017"
```

DOCKER COMPOSE

specify docker-compose version

```
version: '2'
```

Define the services/containers to be run

```
services:
  angular:
....
  express:
.....
  database:
.........
```

DOCKER COMPOSE

- **Specify the directory of the Dockerfile**
- **Specify port forwarding**

```
angular:
    build: angular-client
    ports:
      - "4200:4200"
```

- **Specify the directory of the express service *'express-server'***
- **Specify port forwarding**
- **Link the service with the database service**

```
express:
    build: express-server
    ports:
      - ”3000:3000"
       links:
            - database
```

DOCKER COMPOSE

- **Specify the Image to build the container from**
- **Specify port forwarding**

```
database:
  image: mongo
  ports:
    - "27017:27017"
```

DOCKER COMPOSE COMMANDS

- Entrypoint
- Command
- links
- volume
- ports
- restart
- build

DOCKER COMPOSE LIMITATIONS

What is Docker Compose For?

- Production Systems? Maybe.
- Limitations
  - One computer only
  - Basic scaling ability
  - No load balancing
- Managing single container systems ✔
- Development Environment ✔
  - Environment looks like production from inside container
  - Can run multi-container systems on laptop
  - Filesystems mapping and auto-reload

DOCKER COMPOSE - SUMMARY

- DOCKER COMPOSE
  - Run multi-container systems
  - Declarative configuration
  - Great for development environments
  - Limited use in production

MANAGING CONTAINERS

MANAGING CONTAINERS

- Docker container commands
  - The basics
  - Interacting with your containers
  - Logs and process information
  - Resource limits
  - Container states and miscellaneous commands
  - Removing containers
- Docker networking and volume

BASICS

```
$ docker container run hello-world
```

```
$ docker container ls -a
```

```
$ docker container rm infallible_davinci
```

```
$ docker container run -d --name nginx-test -p 8080:80 nginx
```

```
$ docker container run --name nginx-foreground -p 9090:80 nginx
```

INTERACTING WITH YOUR CONTAINERS

- Attach
  - The first way of interacting with your running container is to attach to the running process. We still have our nginx-test container running, so let's connect to that by running this command:
  - **$ docker container attach nginx-test**
  - We can start our container back up by running the following:
    - **$ docker container start nginx-test**
  - Let's reattach to our process, but this time pass an additional option:
    - **$ docker container attach --sig-proxy=false nginx-test**
- **exec**
  - The attach command is useful if you need to connect to the process your container is running, but what if you need something a little more interactive? You can use the exec command; this spawns a second process within the container that you can interact with.
  - For example, to see the contents of the file /etc/debian_version, we can run the following command:
  - **$ docker container exec nginx-test cat /etc/debian_version**
  - **$ docker container exec -i -t nginx-test /bin/bash**

LOGS & PROCESS INFORMATION

- logs
  - **$ docker container logs --tail 5 nginx-test**
  - To view the logs in real time, I simply need to run the following:
  - **$ docker container logs -f nginx-test**
  - **$ docker container logs --since 2017-06-24T15:00 nginx-test**
  - The logs command shows the timestamps of stdout recorded by Docker and not the time within the container. You can see this from when I run the following commands:
  - **$ date**
    $ docker container exec nginx-test date
  - Luckily, to save confusion--or add to it, depending on how you look at it--you can add -t to your logs commands:
  - **$ docker container logs --since 2017-06-24T15:00 -t nginx-test**

TOP

- top
  - The top command is quite a simple one; it lists the processes running within the container you specify:
  - **$ docker container top nginx-test**

STATS

- The stats command provides real-time information on either the specified container or, if you don't pass a container name or ID, all running containers:
- **$ docker container stats nginx-test**

RESOURCE LIMITS

- The last command we ran showed us the resource utilization of our containers; by default, a container when launched will be allowed to consume all the available resources on the host machine if it requires it. We can put caps on the resources our containers can consume; let's start by updating the resource allowances of our nginx-test container.
- Typically, we would have set the limits when we launched our container using the run command; for example, to halve the CPU priority and set a memory limit of 128M, we would have used the following command:
- **$ docker container run -d --name nginx-test --cpu-shares 512 --memory 128M -p 8080:80 nginx**

- However, we didn't need to update our already running container; to do this, we can use the update command. Now you must have thought that this should entail just running the following command:
- **$ docker container update --cpu-shares 512 --memory 128M nginx-test**

CONTAINER STATES AND MISCELLANEOUS COMMANDS

- $ for i in {1..5}; do docker container run -d --name nginx$(printf "$i") nginx; done

PAUSE & UNPAUSE CONTAINER

- $ docker container pause nginx1
- $ docker container unpause nginx1

STOP, START, RESTART AND KILL

- **$ docker container stop nginx2**
- For example, running the following command will wait up to 60 seconds before sending a SIGKILL, should it need to be sent to kill the process:
  - **$ docker container stop -t 60 nginx3**
- **$ docker container start nginx2 nginx3**
- The restart command is a combination of the following two commands; it stops and then starts the container ID or name you pass it. Also, like stop, you can pass the -t flag:
  - **$ docker container restart -t 60 nginx4**
- **$ docker container kill nginx5**

REMOVING CONTAINERS

- To remove the two exited containers, I can simply run the prune command:
  - **$ docker container prune**
- You can choose which container you want to remove using the rm command; like this, for example:
  - **$ docker container rm nginx4**
- **$ docker container stop nginx3 && docker container rm nginx3**

MISCELLANEOUS COMMANDS

- The create command is pretty similar to the run command, except that it does not start the container, but instead prepares
  - **$ docker container create --name nginx-test -p 8080:80 nginx**
- The next command we are going to quickly look at is the port command; this displays the port along with any port mappings for the container:
  - **$ docker container port nginx-test**
- Before we run the command, let's create a blank file within the nginx-test container using the exec command:
  - **$ docker container exec nginx-test touch /tmp/testing**
  - Now that we have a file called testing in /tmp, we can view the differences between the original image and the running container using the following:
  - **$ docker container diff nginx-test**